1·0   2·8   2·5

1·1   3·15   2·2

       3·5   2·0

       4·0

       4·5   1·8

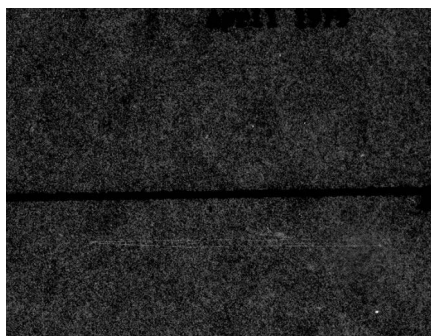1·25   1·4   1·6

NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

14   CMU-CS-79-115

6 SYSTOLIC PRIORITY QUEUES

10   Charles E. Leiserson

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

11   April 1979

9 | *Interim rept.,*

12 | *18p.*

D D C
RECEIVED
OCT 1 1979
B

√403 081

1. Introduction

Very large scale integrated (VLSI) circuit technology has made it possible to build multiprocessor hardware devices to aid in the rapid solution of sophisticated problems. An algorithms designer wishing to take full advantage of the massive parallelism offered by VLSI must address geometric issues hitherto relegated to layout artists. The reason for this is that VLSI is a planar technology in which the interconnections among components on a chip may cost more than the components themselves. The designer of a multiprocessor algorithm to be implemented in this technology must consider the complexity of the data paths between processors in evaluating the algorithm.

Many programming applications require the ability to insert *records* into a set, and at any time to retrieve from the set the record having the smallest *key* according to some ordering. A data structure that provides such services is called a priority queue. (See Knuth [1973], pp. 150-152 and Aho, Hopcroft, and Ullman [1974], pp. 147-152.) The operation INSERT(Q,a) replaces the set Q with the set Q ∪ {a}. The operation EXTRACT_MIN(Q) returns the smallest element a of Q and replaces Q with Q - {a}. This paper shows how high-performance priority queues can be built using the VLSI technology.

Section 2 of this paper discusses systolic systems, the model of parallel computation used for this work. Section 3 presents a systolic array implementation of a priority queue. Section 4 shows how multiple priority queues can be implemented as a single device that shares processors among the queues. The organization of the shared structure is presented in Section 5. Section 6 deals with the geometric layout of the multiple queue device in VLSI. The conclusion is presented in Section 7.

2. Systolic Systems

A systolic system is a network of processors that rhythmically compute and pass data among themselves. The analogy is to the rhythmic contraction of the heart which pulses blood through the circulatory system of the body. Each processor in a systolic network can be thought of as a heart that pumps multiple streams of data through itself. The regular beating of these parallel processors keeps up a constant flow of data throughout the entire network. As a processor pumps data items through, it performs some constant-time computation and may update some of the items.

Systolic systems provide a realistic model of computation which captures the concepts of pipelining, parallelism, and interconnection structures. Kung and Leiserson [1978] demonstrates that many basic matrix computations can be performed by systolic systems whose underlying network is array structured. These systolic arrays are suitable for implementation as VLSI hardware devices. This paper will show the utility of systolic trees.

Unlike the closed-loop circulatory system of the body, a systolic computing system usually has ports into which inputs flow, and ports from which the results are retrieved. Thus a systolic system can be a pipelined system - input and output occur with every pulsation. This makes them attractive as peripheral processors attached to the data channel of a host computer. Figure 1 illustrates how a special-purpose systolic device might form a part of a PDP-11 system. A systolic system might be attached directly to the CPU of a Von Neumann machine, much as a floating-point processor may be added to extend the instruction set of a computer.
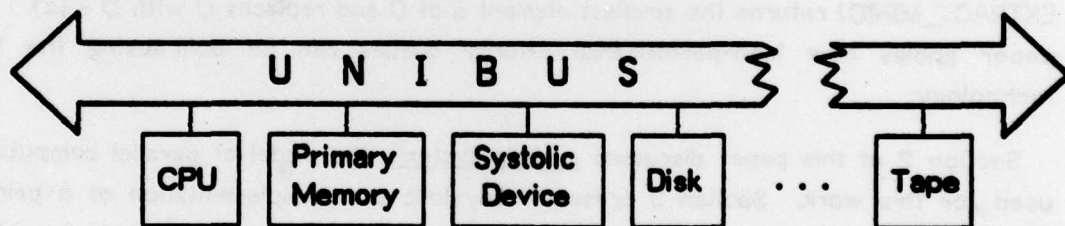
Figure 1: A systolic device connected to the UNIBUS of a PDP-11.

The activities of the processors in a systolic system can be assumed to be synchronous. With each pulse of a clock, a processor executes the same constant-time program. Furthermore, each processor is only allowed a fixed number of input and output lines and a constant amount of local storage. It is possible to view the processors as being asynchronous, each computing its output values when all its inputs are available, as in the data flow model. For the results of this paper, the synchronous approach is more direct and intuitive.

3. A Simple Systolic Priority Queue[*]

A linear systolic array can implement a fast priority queue. Each processor in this array has two registers A and B, and each processor can access the registers of its two neighbors, as shown in Figure 2. The A registers hold elements in the queue in sorted order, with the smallest element in A_1. The B registers contain elements that are being inserted into the queue. Initially, all the elements in the queue are $+\infty$. The priority queue operations INSERT and EXTRACT_MIN are performed by the user at the left end in the diagram. As items are inserted by the host, they displace overflow elements which are output at the right end. Normally, the overflow element will be $+\infty$, but when this is not the case, a real overflow has occurred.

Host Computer { B_0 — B_1 — B_2 — B_3 — B_4 — B_5 — B_6 — B_7 — Overflow
A_0 — A_1 — A_2 — A_3 — A_4 — A_5 — A_6 — A_7 }
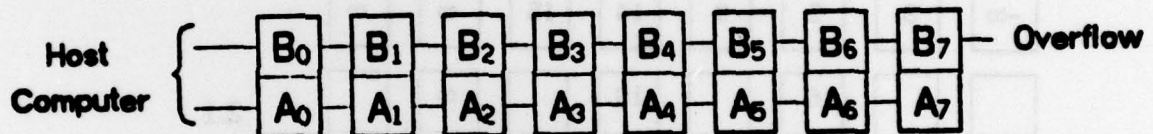
Figure 2: A simple systolic priority queue.

Even and odd numbered processors alternately pulsate, each time executing the following:

1. $B_i \leftarrow B_{i-1}$.

2. Arrange the elements in A_{i-1}, A_i, and B_i so that $A_{i-1} \leq A_i \leq B_i$.

Processor 0 is a dummy processor which does not execute any code, but whose registers can be altered by the host machine. The array pulses twice each time an operation is performed by the host machine, once for odd numbered processors and once for those with even positions. The operation INSERT(Q,a) is implemented by placing the item a in B_0 and $-\infty$ in A_0 just before processor 1 pulses. Each element travels to the right until it finds its place in the array.

By loading A_0 and B_0 with $+\infty$, the pulsation of the systolic array causes

---

[*]This section describes research done jointly with H.T. Kung.

Figure 3: Several steps in the execution of the systolic array
shown in Figure 2.

EXTRACT_MIN to be performed, the minimum value being found in A_0. With each pair of pulsations the systolic array is ready to execute another INSERT or EXTRACT_MIN operation. Figure 3 shows several steps in the execution of this systolic array. The initial configuration in the figure shows the insertion of items 9 and 15 already in progress. Although it may take an element a long time to find its place in the systolic array, to the host computer an INSERT operation appears to take only constant time. Since the minimum element in the queue is always at the front, an EXTRACT_MIN operation also appears to take constant time. The operation of the systolic array is pipelined so that no degradation occurs even when the host executes many priority requests in a row. Thus we may say that the systolic array has a _response time_ which is a constant, independent of the length of the array.

4. The Systolic Multiqueue

Suppose several of the simple priority queues in Section 3 are attached as a device to a host computer. No matter how a fixed number of processors are allocated, the capacity of any particular queue may be exceeded while most of the other queues are empty. In this section a single device is presented that is capable of implementing many priority queues that dynamically share processors. Like the simple queue in the previous section, the systolic multiqueue can perform INSERT and EXTRACT_MIN for a single host computer, on any of m queues, with a response time that is a constant, independent of the size of the queue.

Figure 4 illustrates the organization of the systolic multiqueue. Each of the m queues to be implemented requires a systolic array of the type presented in Section 3. These can be accessed directly by the host computer. When a systolic array overflows, the overflow element travels through a switching network to a large systolic tree. Each time the minimum of a particular queue is extracted from the corresponding systolic array, the minimum of the elements that have overflowed from that queue is removed from the systolic tree. The internal structure of this shared overflow area is examined in Section 5. Here, we only need to know its behavior.

The records stored in the systolic tree are the same as those in the systolic arrays, but an additional field is used to identify the queue from which the item originated. Thus items are stored in the systolic tree according to a composite record $<Q,a>$ where a was originally inserted by an INSERT(Q,a) operation and eventually overflowed from the systolic array corresponding to that queue.
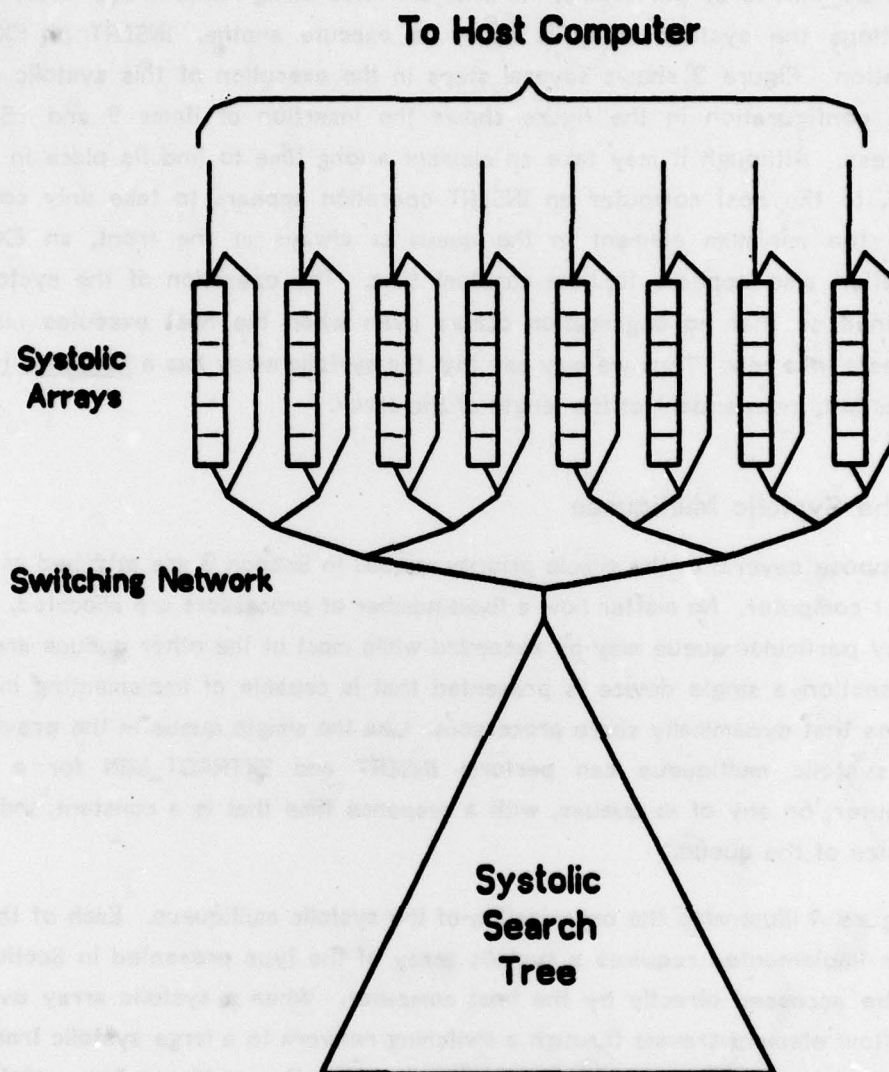
Figure 4: The systolic multiqueue device.

The operations that can be performed by the systolic tree are very similar to those commands given to the entire systolic multiqueue by the host. The composite record <Q,a> is inserted into the systolic tree by INSERT(Q,a), and EXTRACT_MIN(Q) removes the

smallest element in the systolic tree which has Q as its first field. As will be seen in Section 5, a systolic tree of size n can perform each operation in time $O(\log n)$. The operations are pipelined, however, so that the systolic tree can process several operations in parallel, waiting only constant time between successive operations. For example, if a sequence of EXTRACT_MIN's are started constant time apart, it will take $O(\log n)$ for the first minimum to be retrieved, but then results appear with every cycle of the systolic tree. Thus the systolic tree provides high throughput, with $O(\log n)$ response.

The claim was made, however, that the systolic multiqueue had constant response time for each of m queues. Systolic arrays of size proportional to $\log n + \log m$ are used to achieve this goal by satisfying any immediate requests from the host. When the host executes EXTRACT_MIN(Q), that operation is performed on the corresponding systolic array. At the same time, a request is put into the systolic tree to perform an EXTRACT_MIN(Q). A result is yielded by the systolic tree $O(\log n)$ time later and takes $O(\log m)$ more time to traverse the switching network. It is then inserted into the systolic array at the same end the host computer uses. Even if the host has performed $\log n + \log m$ EXTRACT_MIN(Q) operations in the meantime, the quick response systolic array has been able to satisfy the requests. Now if the host continues to perform EXTRACT_MIN(Q)'s, a stream of results from the systolic tree will be inserted into the array just in time to satisfy the requests. The systolic array will always have at least one item in it because operations on the systolic tree are pipelined. It does not matter whether or not the host accesses different queues. Since it can only access one queue at a time, no systolic array will empty before the beginning of a stream of items from the systolic tree has reached the systolic array.

The number of processors in the systolic arrays is $m \log n$. If the size of the systolic tree is doubled, this means only m more processors need be added to the systolic arrays. The amount of sharing of processors among the m queues is clearly substantial. Furthermore, the systolic multiqueue will not overflow until the shared systolic tree overflows. In fact, overflow of the systolic tree can be handled "nicely" as will be seen in Section 5.

5. Systolic Trees

It seems natural to use a tree-structured hardware device to achieve pipelined performance with $O(\log n)$ response time for INSERT(Q,a) and EXTRACT_MIN(Q). After all,

a software implementation on a sequential machine can guarantee $O(\log n)$ performance by using a height-balanced <u>binary</u> <u>search</u> <u>tree</u>. AVL trees, 2-3 trees, and B-trees are popular data structures which have this performance. For a sequential implementation of a single priority queue, a heap is an attractive data structure because heap storage can be managed as easily as stack storage. (Aho, Hopcroft, and Ullman [1974] has a good presentation of several of these techniques.) Unlike most programmed implementations of priority queues, however, the parallel structure required by the systolic multiqueue cannot use a separate data structure for each queue.

A major problem in the design of a hardware search tree is that the standard balanced tree schemes do not map well onto a fixed interconnection structure. A sequential algorithm can move the tree pointers to maintain the balance of the tree. Data usually remains in fixed locations. Since the "pointers" in a hardware tree are electrical wires, data must be moved to maintain the balance of the tree.

Because the systolic multiqueue requires the operations $INSERT(Q,a)$ and $EXTRACT\_MIN(Q)$, keys are considered to be from a composite record $<Q,a>$. A dummy queue number $+\infty$ is used to indicate an empty record. Records are compared by lexicographic ordering, that is, $<Q,a> < <Q',a'>$ if $Q < Q'$ or if $Q = Q'$ and $key(a) < key(a')$.

It is useful to view all operations on the tree as occurring in pairs $[EXTRACT\_MIN(Q'), INSERT(Q,a)]$. Normally, the paired operation involves the dummy queue $+\infty$. For example, when an insertion is performed on an arbitrary queue, a $+\infty$ record is deleted by $EXTRACT\_MIN(+\infty)$. If the systolic tree overflows from too many insertions, however, this exceptional condition can be handled by the operating system of the host computer. The job using a particular queue can be disabled and the elements in that queue can be removed. When an EXTRACT_MIN frees up some space, the elements of that queue can be "swapped" back in by the paired INSERT. The analogy to a virtual memory computer which has a swapping drum is a good one. Queues can be managed just like any other operating system resource. A small amount of bookkeeping is required to keep for each queue, the number of items in the tree.

One scheme for implementing the paired operations is illustrated in Figure 5. Each processor in a systolic array is also a leaf of a systolic tree. A processor P_i contains one record $<Q_i,a_i>$. The tree serves to broadcast paired operations to the processors and to retrieve the EXTRACT_MIN results from the processors. A paired operation $[EXTRACT\_MIN(Q'), INSERT(Q,a)]$ will reach all the processors at the same time. Each
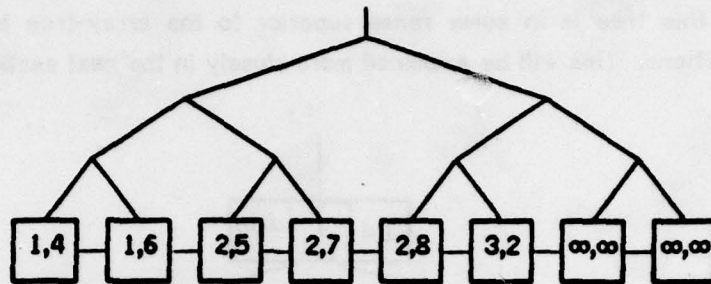
Figure 5: The systolic array-tree.

processor P_i executes:

1. <u>Extraction.</u> If $Q_i = Q'$ and $Q_{i-1} < Q'$, then send $<Q_i,a_i>$ up the tree as the result of EXTRACT_MIN(Q').

2. <u>Left</u> <u>shift.</u> If $Q_{i-1} \geq Q'$, then shift $<Q_i,a_i>$ left to P_{i-1}.

3. <u>Right</u> <u>shift.</u> If $<Q_i,a_i> > <Q,a>$, then shift $<Q_i,a_i>$ right to P_{i+1}.

4. <u>Insertion.</u> If $<Q_{i-1},a_{i-1} \leq <Q,a> < <Q_i,a_i>$, then P_i gets $<Q,a>$.

During the first step, each processor checks to see whether it contains the item to be extracted. After that item is sent on its way up the tree, the elements to the right slide left to take up the empty slot. Then the position for the insertion is determined, and the elements to the right of that processor slide right to make room. Finally, the item to be inserted is placed in the slot left for it. Naturally, the shifts can be optimized so that those elements that slide both left and right do not actually have to move.

Whereas the array-tree keeps all the data at the leaves of the tree, the systolic tree shown in Figure 6 keeps the data in the internal nodes. Consequently, the structure is more like a standard search tree. The processor at each node holds two records, and has connections to its father and two sons. A depth-first tree traversal that prints the left record, recursively visits the left son, recursively visits the right son, and then prints the right record will print out the values in lexicographic order. There is a good reason for having the pointers between the records rather than the normal search tree method of a record between pointers. A balancing similar to the shift step in the systolic array-tree can be performed top-down to permit pipelining of the paired operations.

Each processor need only look at itself and its two sons to determine the shift. The topology of this tree is in some sense superior to the array-tree because there are fewer connections. This will be examined more closely in the next section.
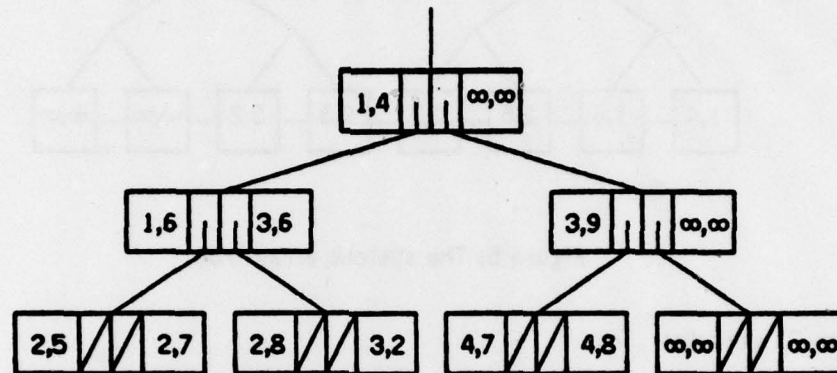

Figure 6: The systolic search tree.

6. VLSI Geometry of the Systolic Multiqueue

Simple and regular interconnections in a VLSI design lead to cheap implementations and high densities. Communication is costly in VLSI, and as the technology improves, the time and energy required for communication grows in comparison with that needed for processing. Therefore, the geometry of the systolic multiqueue must be considered in evaluating the cost of a VLSI implementation.

The linearly connected systolic array easily satisfies the requirement of having a simple geometric realization. The number of external data paths is small as well, emanating only from the ends of the structure. As was shown in Kung and Leiserson [1978], linearly connected systolic arrays are ideal for implementation in VLSI.

More interesting are the structures of the systolic binary trees. Mead and Rem [1978] substantiates the assumption that communication information from the leaf of a VLSI tree to the root takes time proportional to the height of the tree. The fact that the root is the only off-chip connection is highly desirable for VLSI where the number of pins on an IC package is a severe constraint.

The topology of a tree makes a planar embedding easy. In a technology where interconnections frequently occupy much of a chip, the simple connections of a binary tree leave more room for processing elements. It is easy to embed a binary tree in the plane using $O(n \log n)$ area for an n node tree. Figure 7 shows a geometry which realizes this bound. In fact, it is possible to embed a binary tree in the plane using area that is only linear in the number of nodes. This embedding is shown in Figure 8.

Figure 7: Embedding a binary tree in area $O(n \log n)$.

Figure 8: Embedding of a binary tree in linear area.

Whereas the systolic search tree presented in Section 5 can use either geometry, routing the linear connections in Figure 8 appears to be more complex. The tree part of the array-tree is used only for broadcasting, however, and a linear ordering of the leaves need not be the natural ordering shown in Figure 5. This makes the problem simpler, and leads to the linear area geometry of Figure 9.
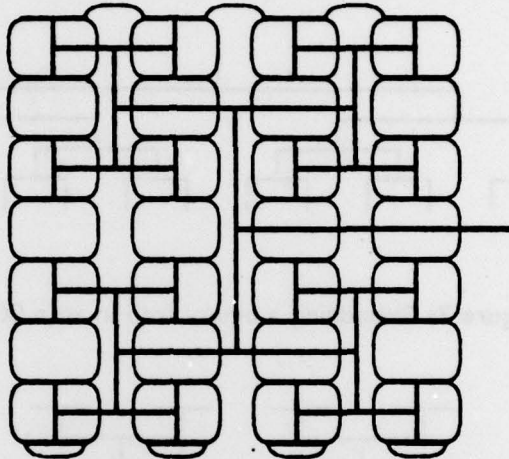

Figure 9: The systolic array-tree embedded in linear area.

There is an advantage to the geometry shown in Figure 7 over that in Figure 8. The linear area solution does not permit connections between the leaves of the tree and the edge of the chip. Although the systolic tree in the systolic multiqueue does not need connections to the leaves, the $O(n \log n)$ area embedding can be used to make a chip that will permit a larger systolic tree to be built up from several chips based on the linear area embedding. Figure 10 shows how this might be done. The decomposition can be very efficient since the number of linear area chips dwarfs the number of $O(n \log n)$ area chips.

7. Conclusion

The systolic multiqueue can be attached to a traditional computer system just like any other device. Because each operation on a queue takes constant time, however, it is
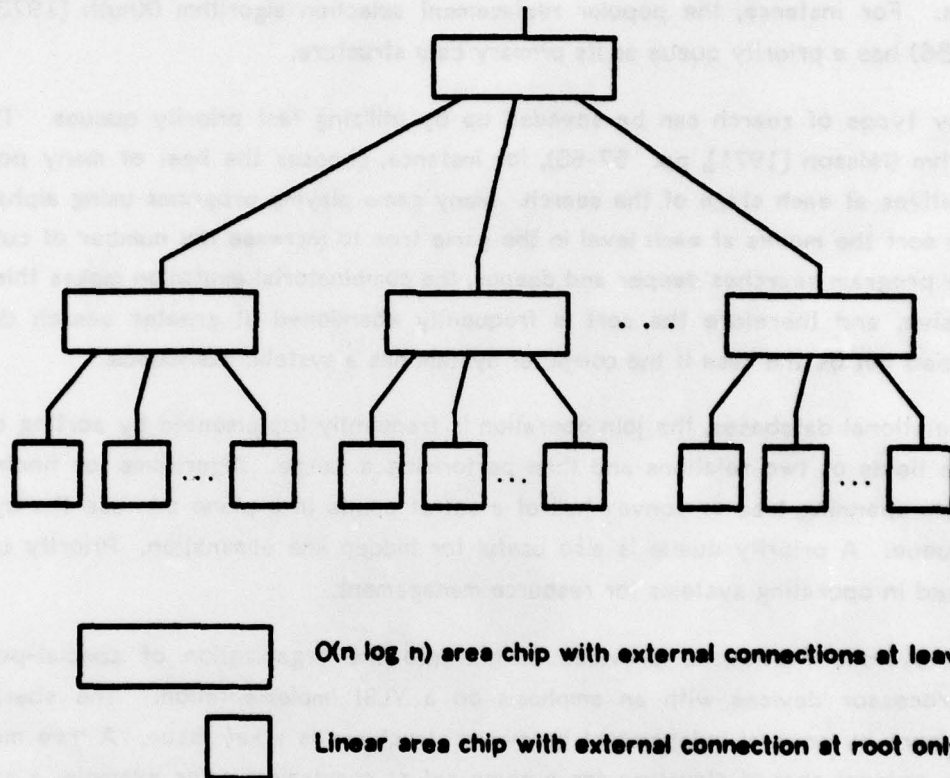
☐ O(n log n) area chip with external connections at leaves.

☐ Linear area chip with external connection at root only.

Figure 10: A large systolic tree as several VLSI chips.

reasonable to connect it directly to the CPU, and make the device visible to the user by extending the instruction set of the computer. Thus the INSERT operation might be a three operand instruction taking a queue number, a key, and a pointer to data. In a multiprogramming or timesharing environment there might be many users of the systolic multiqueue at the same time.

A priority queue is not an obscure data structure, and the uses of priority queues are many. The computation time of sorting alone is sufficient to justify the systolic multiqueue. Internal sorting normally takes $O(n \log n)$ time, but using the systolic multiqueue, we save a factor of $\log n$. Just insert the n items into one of the queues, and then execute n EXTRACT_MIN's. Not only does the computation take less time, but the

load on the CPU of the host machine is lessened. External sorts frequently use priority queues. For instance, the popular replacement selection algorithm (Knuth [1973], pp. 251-256) has a priority queue as its primary data structure.

Many types of search can be speeded up by utilizing fast priority queues. The A* algorithm (Nilsson [1971], pp. 57-65), for instance, chooses the best of many possible alternatives at each stage of the search. Many game playing programs using alpha-beta search sort the moves at each level in the game tree to increase the number of cut-offs. As the program searches deeper and deeper, the combinatorial explosion makes this very expensive, and therefore the sort is frequently abandoned at greater search depths. This need not be the case if the computer system has a systolic multiqueue.

In relational databases, the join operation is frequently implemented by sorting on the chosen fields of two relations and then performing a merge. Algorithms for finding the minimum spanning tree or convex hull of a set of points in a plane can use the systolic multiqueue. A priority queue is also useful for hidden line elimination. Priority queues are used in operating systems for resource management.

The systolic multiqueue provides insight into the organization of special-purpose multiprocessor devices with an emphasis on a VLSI implementation. The sharing of processors by several independent hardware structures is a key issue. A tree may not be the optimal shared structure for a given set of constraints. For example, a systolic array structure that performs like a Young tableau (Knuth [1973], pp. 48-72) might be better under certain conditions, although asymtotically the number of processors dedicated to a single queue grows as the square root of the number of shared processors rather than the logarithm.

The systolic multiqueue can be optimized and modified in many ways. For instance, it is easy to convert the systolic multiqueue into a systolic multideque that implements the priority deque operations INSERT, EXTRACT_MIN, and EXTRACT_MAX. Sten Andler has observed that because only one systolic array in the systolic multiqueue need operate at a time, the m systolic arrays of length $O(\log n)$ might be implemented using only $O(\log n)$ processors each having enough memory to hold m items. Various modifications can be made to the broadcast tree in the systolic array-tree and to the switching network in the systolic multiqueue.

Advances in microelectronics have made the realization of "smart" data structures a

practical reality. VLSI gives us the capability of building logic-in-memory hardware that will drastically change how things are computed. Models of computation based solely on the Von Neumann architecture will be insufficient to evaluate algorithms. Multiprocessor devices like the systolic multiqueue will introduce new cost functions to the sequential algorithm designer. But much work must be done to define and examine the models of parallel computation that lie between the mathematical world of computable functions and the physical world of space and time.

References

Aho, Hopcroft, and Ullman [1974] Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Massachusetts.

Knuth [1973] Knuth, Donald E., Sorting and Searching, Addison-Wesley, Reading, Massachusetts.

Kung and Leiserson [1978] Kung, H. T. and Charles E. Leiserson, "Systolic Arrays (for VLSI)," in Mead and Conway [1978].

Mead and Conway [1978] Mead, Carver A. and Lynn A. Conway, Introduction to VLSI Systems, to be published.

Mead and Rem [1978] Mead, Carver A. and Martin Rem, "Highly Concurrent Structures with Global Communication," in Mead and Conway [1978].

Nilsson [1971] Nilsson, Nils J., Problem Solving Methods in Artificial Intelligence, McGraw-Hill, New York.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-79-115	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SYSTOLIC PRIORITY QUEUES		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Charles E. Leiserson		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE April 1979
		13. NUMBER OF PAGES 18
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1473   EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73
S/N 0102-014-6601 |

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)